Chapter 7 Task Interface Definition

Once tasks in a concurrent design are determined then the subset of data and event flows exchanged among the tasks can be identified and can then be mapped to specific communications between pairs of tasks. This goal is accomplished by applying knowledge about how to define task interfaces. This Task Interface Definition Knowledge is organized as six, distinct, decision-making processes, shown in Figure 26. The first decision-making process, Allocate External Interfaces, maps timers, interrupts, and input and output data from a data/control flow diagram into appropriate interface elements for each task. The process goes on to identify which event flows and data flows are exchanged between tasks in the design. The next two decision-making processes, Allocate Control and Event Flows and Allocate Data Flows, map event flows and data flows, respectively, to appropriate messages and software signals exchanged between pairs of tasks. The fourth decision-making process, Elicit Message Priorities, decides whether to consult with the designer about varying message priorities. The fifth process, Allocate Queue Interfaces, determines what message queuing mechanisms are appropriate for any queued messages in the design. The final process simply allows the designer to review, and, optionally, to assign names to new design elements. The main

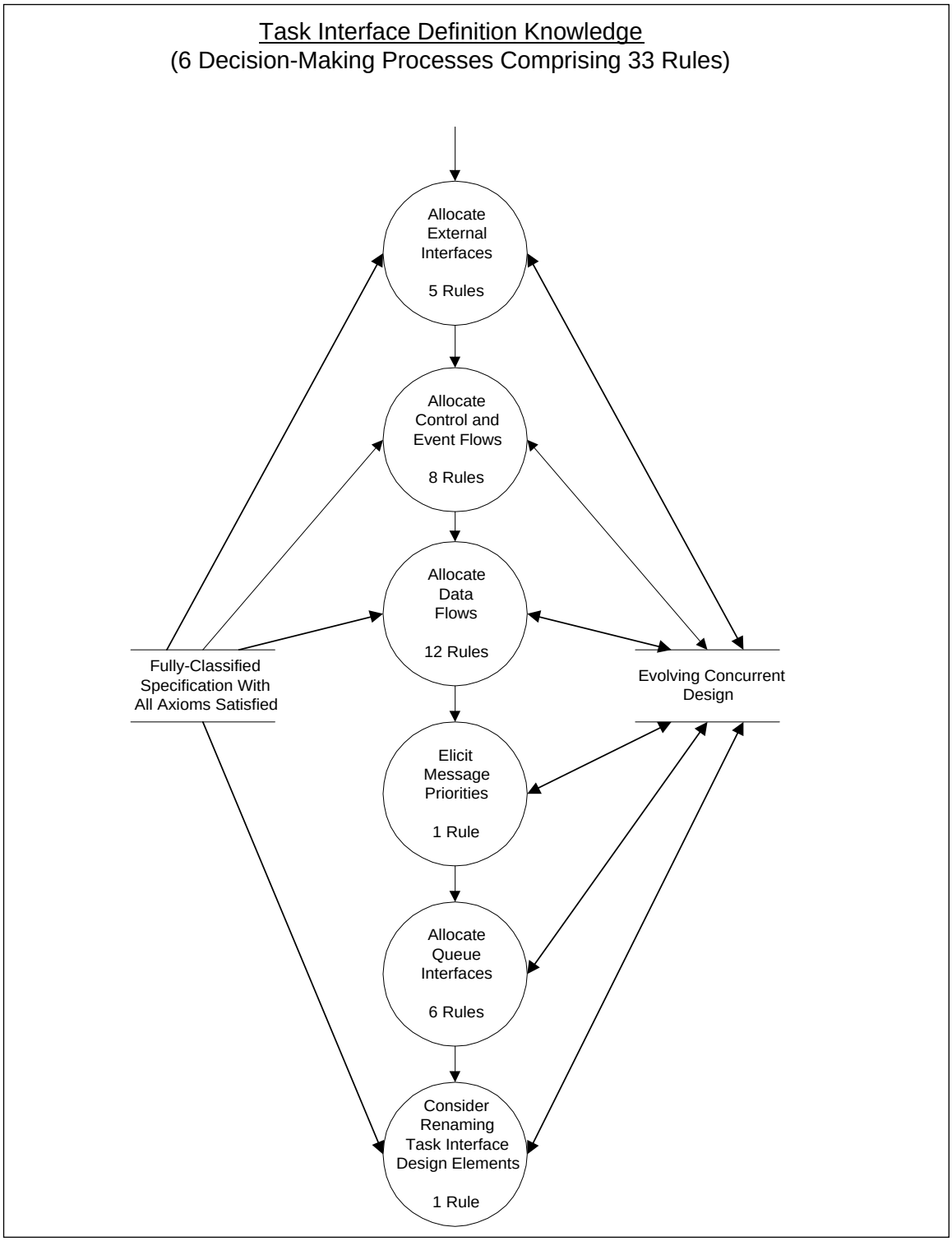

Figure 26. Organization of Task Interface Definition Knowledge

information output from the task-interface definition consists of inter-task messages, including message data, and any message queue interfaces that are required.

The strategy adopted to define task interfaces begins by examining inputs, outputs, timers, and interrupts on the data/control flow diagram. Each input and each output is mapped in the evolving design to data read and written by the appropriate tasks; timers are mapped to timer expiration events and interrupts are mapped to external events for the appropriate tasks. The strategy continues by determining the control and event flows and data flows exchanged between tasks. Subsequently the control and event flows are mapped to software signals, tightly-coupled messages, or queued messages, as appropriate. Design-decision rules are used to recognize specific situations where each type of interface mapping is suitable. In addition, the design-decision rules recognize when a task interface should be referred to an experienced designer because additional information is required. If no experienced designer is available, then default decisions are taken. A similar set of design-decision rules then consider how to map inter-task data flows to appropriate message interfaces, as either tightly-coupled messages or queued messages. Once all the inter-task exchanges are mapped to specific message types, an experienced designer is then given the opportunity to specify differing priorities for queued messages that are received by any task from multiple sources. If no experienced designer is available, then all queued messages are maintained at a single priority. Following the assignment, if any, of priorities to queued messages, the message queuing facilities of the target environment are analyzed and an appropriate queue interface is

defined for each task that receives queued messages. After the mapping of task interfaces is completed, the designer is offered an opportunity to review the new design elements and to assign names to each of them. Each decision-making process within the Task Interface Definition Knowledge base consists of a set of design-decision rules. These rules are specified below.

7.1 Allocate External Task Interfaces

Five rules compose the process that allocates external task interfaces. Each rule is specified in turn below.

7.1.1 Rule to Allocate Timer Events

One design-decision rule allocates a timer event for each periodicity[1] associated with each task. The rule associated with timer allocation is derived from a generalization of the guidance given in the CODARTS design method for allocating timers to periodic tasks. [Gomaa93, pp. 213-214 and pp. 240-241] The rule is specified as follows.

Rule: Timer Event

if $Task_{PERIODIC}$ is derived from Solid $Transformation_{ST}$ and
$Solid\ Transformation_{ST}$ is the sink for a $Timer_T$ and
$Timer_T$ has a $Period_P$
then if $Task_{PERIODIC}$ does not already receive an $Event_{TE}$ of type Timer
with an interval equal to $Period_P$
then create an $Event_{TE}$ of type Timer with an interval equal to $Period_P$
establish the design relationship $Task_{PERIODIC}$ Accepts $Event_{TE}$
record the decision and rationale in the design history for $Event_{TE}$
else use existing $Event_{TE}$
fi
denote the traceability between $Timer_T$ and $Event_{TE}$
fi

[1] Remember that a given task can, where required, operate with different periods during the execution of a real-time application.

An example where this rule applies can be found in the cruise control and monitoring system case study presented by Gomaa. [Gomaa93, Chapter 22] In the example, several tasks are derived from periodic functions, including Monitor Auto Sensors, Perform Calibration, Determine Speed and Distance, Throttle, Check Maintenance Needed, Monitor Reset Buttons, and Compute Average Mileage. The rule specified above will generate timer events for each of these tasks.

7.1.2 Rule to Allocate External Events

A similar rule generates an external event for each interrupt that is received by each task in the evolving design. This rule is derived from the CODARTS guidance for generating external events based on interrupts that activate an asynchronous task. [Gomaa93, p. 214] The rule is specified as follows.

Rule: Interrupt

if

$Task_{ASYNCHRONOUS}$ is derived from Interface $Object_{IO}$ and
Interface $Object_{IO}$ is the sink for an $Interrupt_{I}$

then

create an $Event_{EE}$ of type External
establish the design relationship $Task_{ASYNCHRONOUS}$ Accepts $Event_{EE}$
record the decision and rationale in the design history for $Event_{EE}$
denote the traceability between $Interrupt_{I}$ and $Event_{EE}$

fi

An example where this rule applies can be found in the cruise control and monitoring system case study. In this example, two tasks, Monitor Shaft Rotation and Monitor Cruise Control Input, each service an asynchronous device. Each of these tasks

processes interrupts for the appropriate device. The rule specified above will map these interrupts to an external event.

7.1.3 Rules for Allocating Data

The two additional task interface rules deal with input and output data, rather than task activation. Each input from the data/control flow diagram is mapped to a data input for appropriate tasks, according to the rule below.

Rule: Data Input

if
$Task_{ANY}$ is derived from Interface $Object_{IO}$ and
Interface $Object_{IO}$ is the sink for an $Input_{I}$
then
create a $Data_{IN}$
establish the design relationship $Task_{ANY}$ Reads $Data_{IN}$
record the decision and rationale in the design history for $Data_{IN}$
denote the traceability between $Input_{I}$ and $Data_{IN}$
fi

Similarly, each output from the data/control flow diagram is mapped to a data output for appropriate tasks, according to the rule below.

Rule: Data Output

if
$Task_{ANY}$ is derived from Interface $Object_{IO}$ and
Interface $Object_{IO}$ is the source for an $Output_{O}$
then
create a $Data_{OUT}$
establish the design relationship $Task_{ANY}$ Writes $Data_{OUT}$
record the decision and rationale in the design history for $Data_{OUT}$
denote the traceability between $Output_{O}$ and $Data_{OUT}$
fi

These two rules are derived from the general guidance provided by the CODARTS design method when discussing the integration of tasks and modules for asynchronous and polled input/output devices. [Gomaa93, pp. 239-242] In some situations, the designer might choose to combine inputs to or outputs from a task into single inputs or single outputs. For example, in the cruise control and monitoring system described by Gomaa, [Gomaa93, Chapter 22] the two inputs, Brake Input and Engine Input, read by the task Monitor Auto Sensors might be combined into a single input, Sensor Input. In other situations, such as the elevator control system presented by Gomaa, [Gomaa93, Chapter 23] multiple inputs from the elevator motor and elevator door should be kept separate. The approach adopted within the design-decision rules defined in this dissertation is to keep the separate identities of all inputs to and outputs from tasks.

7.1.4 Rule to Identify Inter-Task Exchanges

One design-decision rule identifies and tags each event flow and data flow exchanged between tasks in the evolving design. This rule views the event flows and data flows exchanged among transformations in the light of how those transformations are allocated to tasks in the evolving design. Given such a view, the event flows and data flows exchanged between tasks can be identified with ease. The rule is specified below.

Rule: Inter-Task Exchange

if

Transformation$_{SENDER}$ is allocated to Task$_{SENDER}$ and
Transformation$_{RECEIVER}$ is allocated to Task$_{RECEIVER}$ and
Task$_{SENDER}$ is not Task$_{RECEIVER}$ and
Transformation$_{SENDER}$ is not allocated to Task$_{RECEIVER}$ and
Transformation$_{RECEIVER}$ is not allocated to Task$_{SENDER}$ and
Directed Arc$_{DA}$ flows from Transformation$_{SENDER}$ to
Transformation$_{RECEIVER}$ and
Directed Arc$_{DA}$ is a Control Event Flow or an Internal Data Flow or a
Signal

then

mark Directed Arc$_{DA}$ as an Inter-Task Exchange

fi

Since this rule only performs a clerical function to facilitate later decision making, no example is necessary.

7.2 Allocate Control and Event Flows

The second decision-making process considers how to map control and event flows, that is, dashed, directed arcs, from the data/control flow diagram to inter-task events or messages. First consideration is given to mapping control and event flows to inter-task events and then to mapping control and event flows to tightly-coupled or queued messages. Control and event flows that cannot be mapped to either inter-task events or messages are referred to an experienced designer to elicit additional information or, where no experienced designer is available, a default decision maps these control and event flows to queued messages.

7.2.1 Rules for Mapping to Software Events

When a target environment supports software signals between tasks (see Chapter 5 for the minimum assumptions made regarding software signals), control and event flows can be mapped to inter-task events because no data is transmitted along with them.[2] The CODARTS design method identifies three cases where control and event flows might require mapping to inter-task messages rather than inter-task events. [Gomaa93, pp. 213-217] One case occurs when the source and destination tasks reside on different processors in a distributed system. Since distributed systems are outside the scope of this dissertation, this case is not addressed in the design-decision rules. A second case occurs when a destination task receives several different event flows from the same source task. A third case occurs when a destination task receives event flows from several different source tasks and where: 1) events cannot be missed and 2) the order of arrival must be preserved. The second and third cases are addressed by the design-decision rules that follow.

The first design-decision rule identifies situations where control flows can be mapped to inter-task events. In this context, control flows are defined to be those dashed, directed arcs that emitted from a control object; these can include Triggers, Enables, Disables, and Signals. The rule, specified below, maps each inter-task control flow to an inter-task event, unless the number of events flowing between the two tasks exceeds a

---

[2] Whether inter-task event flows have any advantage over inter-task messages depends on the details of a particular target environment. Where no advantage exists, inter-task event flows can be factored out of a design through the target environment description (see Chapter 5).

threshold contained in the target environment description or unless the sending task is an inverted task. Events flowing to an inverted task must identify their source, and thus data must be included with the event. To send data between tasks a message is necessary.

Rule: Control Flow To Software Event (First Preference)

if
Directed Arc_{CF} flows from a $Task_{SENDER}$ to a $Task_{RECEIVER}$ and
Directed Arc_{CF} is a Signal or Control Event Flow and
the source of Directed Arc_{CF} is a Control Object and
$Task_{SENDER}$ is not inverted and
the number of control flows going from $Task_{SENDER}$ to
$Task_{RECEIVER}$ does not exceed the maximum number of
inter-task signals specified in the target environment description
then
if Directed Arc_{CF} is a Disable and an $Event_{IE}$ of type Internal
with the name Disable already exists from $Task_{SENDER}$ to
$Task_{RECEIVER}$
then use existing $Event_{IE}$
else create an $Event_{IE}$ of type Internal from $Task_{SENDER}$ to $Task_{RECEIVER}$
establish the design relationship $Task_{SENDER}$ Generates $Event_{IE}$
establish the design relationship $Task_{RECEIVER}$ Accepts $Event_{IE}$
fi
denote the traceability between Directed Arc_{CF} and $Event_{IE}$
record the decision and rationale in the history for $Event_{IE}$
fi

An example where this rule might apply appears in a cruise control and monitoring system discussed by Gomaa. [Gomaa93, Chapter 22] In the example, six control flows, three enables and three disables, flow from the task Cruise Control to the task Auto Speed Control. Since each disable simply deactivates the task, the three disables can be compressed to a single, software signal, and, therefore, the number of control flows between the tasks can be counted as four. Where the target environment description

indicates that the threshold for maximum number of inter-task signals is at least four, the rule defined above will map these control flows to inter-task signals. Should the threshold for maximum number of inter-task signals fall below four then the rule defined above will not map the control flows to inter-task events.

A similar rule is defined for event flows, that is, dashed, directed arcs that are not control flows, except that an additional factor is considered. If the receiving task receives event flows from multiple sources, or if the sending task is inverted, then the incoming event flows are not mapped to inter-task events, except where those event flows are locked-state events. Locked-state events can be mapped to inter-task events because the destination task is suspended until the event arrives.

Rule: Event Flow To Software Event (First Preference)

if

$Signal_{EF}$ flows from a $Task_{SENDER}$ to a $Task_{RECEIVER}$ and
the source of $Signal_{EF}$ is not a Control Object and
$Task_{SENDER}$ is not inverted and
($Signal_{EF}$ is a **locked-state event** for a Control Object or
no task other than $Task_{SENDER}$ sends event flows to $Task_{RECEIVER}$) and
the number of event flows going from $Task_{SENDER}$ to
$Task_{RECEIVER}$ does not exceed the maximum number of
inter-task signals specified in the target environment description

then

create an $Event_{IE}$ of type Internal from $Task_{SENDER}$ to $Task_{RECEIVER}$
establish the design relationship $Task_{SENDER}$ Generates $Event_{IE}$
establish the design relationship $Task_{RECEIVER}$ Accepts $Event_{IE}$
denote the traceability between $Signal_{EF}$ and $Event_{IE}$
record the decision and rationale in the history for $Event_{IE}$

fi

Some examples where this rule might apply appear in a robot controller case study elaborated by Gomaa. [Gomaa93, Chapter 23] For this discussion assume that the maximum number of inter-task signals that can be exchanged between tasks is two. In the robot controller example, two event flows, Resume and Stop, flow from the Robot Command Processor task to the Axis Manager task. The rule defined above maps these two events to inter-task signals. Another relevant case appears in the robot controller example where a single event flow, Ended, flows from the Interpreter task to the Robot Command Processor task. Since the Robot Command Processor task also receives event flows from the Control Panel Input Handler, the rule defined above would normally not map the Ended event to an inter-task signal; however, the Ended event is mapped to an inter-task signal because the event is a locked-state event for the Control Robot control object.

7.2.2 Rules for Mapping to Tightly-Coupled Messages

After considering possible mappings of control and event flows to inter-task signals, any control and event flows that could not be mapped can be considered for mapping to tightly-coupled message interfaces. Two rules are defined. The first rule, specified below, maps a control flow from a control object onto a tightly-coupled message.

Rule: Control Flow To Tightly-Coupled Message (Second Preference)

if

$Task_{SENDER}$ sends a Directed Arc_{CF} to $Task_{RECEIVER}$ and
Directed Arc_{CF} is a Signal or Control Event Flow and
the source of Directed Arc_{CF} is a Control Object

then

create a Tightly-Coupled $Message_{CONTROL}$ from $Task_{SENDER}$ to $Task_{RECEIVER}$
establish the design relationship $Task_{SENDER}$ Sends $Message_{CONTROL}$
establish the design relationship $Task_{RECEIVER}$ Receives $Message_{CONTROL}$
denote the traceability between Directed Arc_{CF} and $Message_{CONTROL}$
record the decision and rationale in the history for $Message_{CONTROL}$

fi

Returning to the cruise control and monitoring system discussed in the preceding section, assume that the maximum number of inter-task signals is set at two. In this case the control flows from the Cruise Control task to the Auto Speed Control task would not be mapped to inter-task signals because their number, four, exceeds the allowed threshold in the target environment description. Instead, the second-preference rule, defined above, maps these control flows to a tightly-coupled message.

A similar rule maps locked-state events that could not be mapped to inter-task events to tightly-coupled messages. This rule is specified below.

Rule: Locked-State Event Flow To Tightly-Coupled Message
(Second Preference)

if

$Task_{SENDER}$ sends a $Signal_{EF}$ to $Task_{RECEIVER}$ and
$Task_{RECEIVER}$ is not inverted and
the sink of $Signal_{CF}$ is a $Control\ Object_{CO}$ and
$Signal_{CF}$ is a **locked-state event** for $Control\ Object_{CO}$ and
the sender of $Signal_{CF}$ sends no other Signals to $Control\ Object_{CO}$
unless those other Signals are also **locked-state events** for
$Control\ Object_{CO}$

then

create a Tightly-Coupled $Message_{EVENT}$ from $Task_{SENDER}$ to $Task_{RECEIVER}$
establish the design relationship $Task_{SENDER}$ Sends $Message_{EVENT}$
establish the design relationship $Task_{RECEIVER}$ Receives $Message_{EVENT}$
denote the traceability between $Signal_{EF}$ and $Message_{EVENT}$
record the decision and rationale in the history for $Message_{EVENT}$

fi

An example where this rule might apply can be illustrated by reconsidering the robot controller case study discussed previously (and covered in depth in Appendix C). Assume that the target environment description indicates that no inter-task events are permitted. In this case, the rule defined above would map the Ended event, a locked-state event for the Control Robot control object, to a tightly-coupled message sent from the task Interpreter to the task Robot Command Processor.

7.2.3 Rule for Mapping to Queued Messages

In some situations where they cannot be mapped to inter-task events, event flows might be mapped to queued messages. One specific situation where this mapping can be made involves events that flow to a control task from a device input task. Device input tasks should not be delayed waiting for the control task to accept events because input

events might occur in quick succession. In general then, events from device input tasks to control tasks that cannot be mapped to inter-task events should be mapped to queued messages. An exception to this rule occurs when the events from a device input task arrive at the control task only when the control task is in a locked-state waiting for those events. The rule is specified below.

Rule: Input Event To Queued Message (Second Preference)

if

$Task_{DIT}$ sends a $Signal_{EF}$ to $Task_{CONTROL}$ and
the sink of $Signal_{EF}$ is a Control $Object_{CO}$ and
$Signal_{CF}$ is not a **locked-state event** for Control $Object_{CO}$ and
$Task_{DIT}$ is a periodic or asynchronous device input task and
$Task_{CONTROL}$ is a control task and
the source of $Signal_{EF}$ is a Periodic or Asynchronous Device Input Object

then

create a Queued $Message_{EVENT}$ from $Task_{DIT}$ to $Task_{CONTROL}$
establish the design relationship $Task_{DIT}$ Sends $Message_{EVENT}$
establish the design relationship $Task_{CONTROL}$ Receives $Message_{EVENT}$
denote the traceability between $Signal_{EF}$ and $Message_{EVENT}$
record the decision and rationale in the history for $Message_{EVENT}$

fi

An example where this rule applies can be found in the cruise control and monitoring system discussed previously. In the example, the periodic, device-input task Monitor Auto Sensors sends four event flows to the control task Cruise Control. The rule specified above maps these event flows onto a queued message from Monitor Auto Sensors to Cruise Control.

7.2.4 Rule for Mapping Ambiguous Control and Event Flows

As specified to this point, the rules for mapping control and event flows to inter-task signals or messages recognize several specific situations where a preferred

decision can be taken. Any remaining situations cannot be resolved without additional information, except by taking a default decision. The next rule specified recognizes these less constrained situations, where events flow between tasks in the design, and refers each situation to an experienced designer to elicit additional information that might lead to an appropriate design choice. If an experienced designer is available, then the designer is asked whether the sender of the event must wait until the message is accepted by the receiver before the sender can continue, or whether the sender can continue independently of the receiver accepting the message. Where the sender must wait then the event flow is mapped to a tightly-coupled message; otherwise, the event flow is mapped to a queued message. When an experienced designer is not available or where an experienced designer cannot supply the requested information for a given situation, then the rule maps event flows to a queued message between tasks. The rule is given last preference so that rules recognizing more specific situations take precedence. The rule is specified below.

Rule: Event Flow To Message (Last Preference)

if

$Task_{SENDER}$ sends a $Signal_{EF}$ to $Task_{RECEIVER}$

then

if an experienced designer is available

then show the designer all event flows from $Task_{SENDER}$ to $Task_{RECEIVER}$
ask the designer whether the sender must wait for the receiver to accept these events before continuing

if the designer says the sender need not wait or the designer does not know the answer

then create a Queued $Message_{EVENT}$ from $Task_{SENDER}$ to $Task_{RECEIVER}$
establish the design relationship $Task_{SENDER}$ Sends $Message_{EVENT}$
establish the design relationship $Task_{RECEIVER}$ Receives $Message_{EVENT}$
denote the traceability between $Signal_{EF}$ and $Message_{EVENT}$
record the decision and rationale in the history for $Message_{EVENT}$

else create a Tightly-Coupled $Message_{EVENT}$ from $Task_{SENDER}$ to $Task_{RECEIVER}$
establish the design relationship $Task_{SENDER}$ Sends $Message_{EVENT}$
establish the design relationship $Task_{RECEIVER}$ Receives $Message_{EVENT}$
denote the traceability between $Signal_{EF}$ and $Message_{EVENT}$
record the decision and rationale in the history for $Message_{EVENT}$

fi

else create a Queued $Message_{EVENT}$ from $Task_{SENDER}$ to $Task_{RECEIVER}$
establish the design relationships $Task_{SENDER}$ Sends $Message_{EVENT}$ and $Task_{RECEIVER}$ Receives $Message_{EVENT}$
denote the traceability between $Signal_{EF}$ and $Message_{EVENT}$
record the decision and rationale in the history for $Message_{EVENT}$

fi

fi

An example where this rule applies can be drawn from the same cruise control and monitoring system used previously. An internal task, Auto Speed Control, sends an

event, Reached Cruising, to a control task, Cruise Control. Since this event flow does not correspond to any previously defined situation, no inference maps Reached Cruising to a specific type of message. This case, then, is referred to an experienced designer or mapped to a queued message by default when no experienced designer is available.

7.2.5 Rules for Mapping to Existing Messages

The design rules specified in this dissertation assume that all event exchanges between specific pairs of tasks occur in a uniform manner. So, for example, if a $task_A$ sends one event to another $task_B$ using a queued message, then all events from $task_A$ to $task_B$ will be sent using a queued message. This assumption allows specific inferences to be drawn whenever a mechanism exists to carry any control or event flow between a pair of tasks. Specifically, once a message interface is defined to carry a control or event flow from a source task to a destination task then any remaining control and event flows with the same source and destination can be mapped to the same message interface.

Two rules are specified to perform these mappings. One rule maps control and event flows onto an existing tightly-coupled message between two tasks. The second rule maps control and event flows onto an existing queued message between two tasks. These rules are given first preference because once a specific message interface is defined these additional mappings can be performed immediately. The rule to map control and event flows onto an existing tightly-coupled message is specified below.

Rule: Ride On Existing Tightly-Coupled Message (First Preference)

if

$Task_{SENDER}$ sends a Directed Arc_{EF} to $Task_{RECEIVER}$ and
Directed Arc_{EF} is a Signal or a Trigger or an Enable or a Disable and
$Task_{SENDER}$ sends a Tightly-Coupled $Message_{EM}$ to $Task_{RECEIVER}$

then

if an appropriate Message $Data_{MD}$ does not exist for $Message_{EM}$
then create an appropriate Message $Data_{MD}$ for $Message_{EM}$
establish the design relationship $Message_{EM}$ Includes Message $Data_{MD}$
record the decision and rationale in the history for Message $Data_{MD}$
else use an existing Message $Data_{MD}$ for $Message_{EM}$
fi
denote the traceability between Directed Arc_{EF} and Message $Data_{MD}$
record the decision and rationale in the history for Message $Data_{MD}$

fi

An example where this rule applies can be found in the cruise control and monitoring system discussed previously (and covered in depth in Appendix B). In the example, the task Cruise Control sends a set of control flows to the task Auto Speed Control. Once any one of these control flows is mapped to a tightly-coupled message then the rule specified above will map the remaining control flows in the set to the same tightly-coupled message. The identity of the individual control flows is not lost; each control flow can be distinguished using a data field created for the tightly-coupled message. A similar rule, specified below, maps multiple event flows onto an existing queued message interface for a specific source task and destination task.

Rule: Ride On Existing Queued Message (First Preference)

if
    $Task_{SENDER}$ sends a $Signal_{EF}$ to $Task_{RECEIVER}$ and
    $Task_{SENDER}$ sends a Queued $Message_{EM}$ to $Task_{RECEIVER}$
then
    if an appropriate Message $Data_{MD}$ does not exist for $Message_{EM}$
    then create an appropriate Message $Data_{MD}$ for $Message_{EM}$
        establish the design relationship $Message_{EM}$ Includes Message $Data_{MD}$
        record the decision and rationale in the history for Message $Data_{MD}$
    else use an existing Message $Data_{MD}$ for $Message_{EM}$
    fi
    denote the traceability between $Signal_{EF}$ and Message $Data_{MD}$
    record the decision and rationale in the history for Message $Data_{MD}$
fi

This rule applies in the same cruise control and monitoring system where a task, Monitor Auto Sensors, sends four event flows to the Cruise Control task. Once one of these event flows is mapped by another rule to a queued message, then the rule defined above will map the other three event flows onto the same queued message.

7.3 Allocate Data Flows

The third decision-making process within the Task Interface Definition Knowledge base considers how each inter-task data flow in an evolving design should be mapped to a message interface. The approach adopted is similar to the approach taken when mapping control and event flows. First, when specific situations can be recognized where a particular mapping, either to a tightly-coupled or queued message, appears appropriate then a rule is defined. Second, when a situation is ambiguous then an experienced designer is consulted, where possible, and a default decision is taken when no experienced designer is available. Third, once a specific message type is defined for a

particular source task and destination task then all data flows with the same source and destination are mapped onto that message. The design-decision rules are specified and explained below.

7.3.1 Rules for Mapping to Tightly-Coupled Messages

Two specific situations call for mapping a data flow onto a tightly-coupled message. The first situation occurs when a function within a task receives a data flow from a transformation outside that task and the receiving function generates signals for a control object within the receiving task and all the signals generated by the receiving function are locked-state events for the control object. This rule is specified below.

Rule: Stimulus For Locked-State Event (Second Preference)

if

a $Function_F$ in $Task_{RECEIVER}$ receives a $Stimulus_{DF}$ from a $Task_{SENDER}$ and
$Function_F$ sends a $Signal_{EF}$ to Control $Object_{CO}$ and
$Signal_{EF}$ is a **locked-state event** for Control $Object_{CO}$ and
$Task_{RECEIVER}$ is not inverted

then

create a Tightly-Coupled $Message_{EM}$ from $Task_{SENDER}$ to $Task_{RECEIVER}$
establish the design relationship $Task_{SENDER}$ Sends $Message_{EM}$
establish the design relationship $Task_{RECEIVER}$ Receives $Message_{EM}$
denote the traceability between $Stimulus_{DF}$ and $Message_{EM}$
record the decision and rationale in the history for $Message_{EM}$
create an appropriate Message $Data_{MD}$ for $Message_{EM}$
establish the design relationship $Message_{EM}$ Includes Message $Data_{MD}$
denote the traceability between $Stimulus_{DF}$ and Message $Data_{MD}$
record the decision and rationale in the history for Message $Data_{MD}$

fi

An example where this rule applies can be found in the elevator control system case study discussed previously (and covered in depth in Appendix D). In the example, a

function, Check This Floor, inside the task Elevator Controller receives a data flow, Floor Number, from the task Monitor Arrival Sensors. Check This Floor generates one event, Approaching Requested Floor, that arrives when the control object, Elevator Control, is in a locked-state waiting for that event. Thus, the data flow, Floor Number, will only arrive when the receiving task is awaiting it, so the data flow can be mapped to a tightly-coupled message.

The second situation where a data flow can be mapped to a tightly-coupled message occurs when the data flow is a response. By definition the definition of a Response (see Chapter 4), the receiver of a response is waiting for it and, so, the sender will not be delayed.

Rule: Response To Tightly-Coupled Message (Fourth Preference)

if

$Task_{SENDER}$ sends a $Response_{R}$ to $Task_{RECEIVER}$ and
the sink of $Response_{R}$ is the source of a $Stimulus_{DF}$ and
$Stimulus_{DF}$ is allocated to a $Message_{REQUEST}$

then

create a Tightly-Coupled $Message_{REPLY}$ from $Task_{SENDER}$ to $Task_{RECEIVER}$
establish the design relationship $Task_{SENDER}$ Sends $Message_{REPLY}$
establish the design relationship $Task_{RECEIVER}$ Receives $Message_{REPLY}$
establish the design relationship $Message_{REPLY}$ Answers $Message_{REQUEST}$
denote the traceability between $Response_{R}$ and $Message_{REPLY}$
record the decision and rationale in the history for $Message_{REPLY}$
create an appropriate Message $Data_{MD}$ for $Message_{REPLY}$
establish the design relationship $Message_{REPLY}$ Includes Message $Data_{MD}$
denote the traceability between $Response_{R}$ and Message $Data_{MD}$
record the decision and rationale in the history for Message $Data_{MD}$

fi

An example where this rule applies can be found in a distributed factory automation system described by Gomaa. [Gomaa93, Chapter 25] In the example, a transformation, Production Management, sends a data flow, Process Plan Request, to a transformation, Process Planning, which responds with a data flow, Process Plan Information. Process Plan Information, being classified as a response, would be mapped by the rule specified above to a tightly-coupled message.

7.3.2 Rules for Mapping to Queued Messages

Several specific situations call for mapping a data flow to a queued message. One such situation occurs whenever an data flow is sent to a resource-monitor task or to a control task. A design-decision rule for this situation is specified below.

Rule: Stimulus To Resource-Monitor Or Control Task (Third Preference)

if
  $Task_{SENDER}$ sends a $Stimulus_{DF}$ to $Task_{RECEIVER}$ and
  $Task_{RECEIVER}$ is a resource-monitor or control task
then
  if a $Message_{DFM}$ does not already exist from $Task_{SENDER}$ to $Task_{RECEIVER}$
  then create a Queued $Message_{DFM}$ from $Task_{SENDER}$ to $Task_{RECEIVER}$
    establish the design relationship $Task_{SENDER}$ Sends $Message_{DFM}$
    establish the design relationship $Task_{RECEIVER}$ Receives $Message_{DFM}$
    denote the traceability between $Stimulus_{DF}$ and $Message_{DFM}$
    record the decision and rationale in the history for $Message_{DFM}$
    create an appropriate Message $Data_{MD}$ for $Message_{DFM}$
    establish the design relationship $Message_{DFM}$ Includes Message $Data_{MD}$
    denote the traceability between $Stimulus_{DF}$ and Message $Data_{MD}$
    record the decision and rationale in the history for Message $Data_{MD}$
  else
    denote the traceability between $Stimulus_{DF}$ and $Message_{DFM}$
  fi
fi

An example where this rule applies appears in the elevator control system case study presented by Gomaa. [Gomaa93, Chapter 24] In the example, two resource-monitor tasks, Floor Lamps Monitor and Direction Lamps Monitor, receive data flows from multiple instances of an Elevator Controller task. The rule specified above maps these data flows onto queued messages.

Another situation where data flows between tasks should be mapped to queued messages occurs when the data flows originate from a periodic or asynchronous device input task. In general, the sending task should not be held up waiting for the recipient to accept the data or a subsequent input might be missed. A rule for this situation is specified below

Rule: Stimulus From Device Input Task (Third Preference)

if
Task$_{DIT}$ sends a Stimulus$_{DF}$ to Task$_{RECEIVER}$ and
Task$_{DIT}$ is a periodic or an asynchronous device input task
then
if a Message$_{DFM}$ does not already exist from Task$_{DIT}$ to Task$_{RECEIVER}$
then create a Queued Message$_{DFM}$ from Task$_{DIT}$ to Task$_{RECEIVER}$
establish the design relationship Task$_{DIT}$ Sends Message$_{DFM}$
establish the design relationship Task$_{RECEIVER}$ Receives Message$_{DFM}$
denote the traceability between Stimulus$_{DF}$ and Message$_{DFM}$
record the decision and rationale in the history for Message$_{DFM}$
create an appropriate Message Data$_{MD}$ for Message$_{DFM}$
establish the design relationship Message$_{DFM}$ Includes Message Data$_{MD}$
denote the traceability between Stimulus$_{DF}$ and Message Data$_{MD}$
record the decision and rationale in the history for Message Data$_{MD}$
else
denote the traceability between Stimulus$_{DF}$ and Message$_{DFM}$
fi
fi

An example where this rule applies can be found in the elevator control system case study referred to previously. In the example, an asynchronous device input task, Monitor Floor Buttons, sends a data flow, Service Request, to an internal task, Scheduler. The rule specified above maps this data flow onto a queued message.

A third situation where data flows can be mapped to queued messages arises when a task receives only data flows that are classified as stimuli, but where the task receives such data flows from multiple sources. This condition is represented as a predicate in the following rule. A precise definition of the predicate is given below.

Predicate Definition: **receives only Stimuli from multiple sending tasks**

construct the $Set_{SENDERS}$ of tasks such that each member $Task_T$ sends a
Stimulus to $Task_{RECEIVER}$ and $Task_{RECEIVER}$ does not Accept an
Event Generated by $Task_T$ and $Task_{RECEIVER}$ does not already Receive a
Message Sent by $Task_T$
if the total cardinality over all members in $Set_{SENDERS}$ exceeds one
then return TRUE
else return FALSE
fi

A task that satisfies this predicate is a server task, that is a task that receives data flows from several clients or sources and that handles each data flow in turn. Tasks from which control and event flows are received are excluded from the predicate because such tasks probably have a more complex relationship with the receiving task than that exhibited by a simple client. This more complex relationship might be better handled by some other rule or might warrant referring the interface to an experienced designer. The rule specified to recognize client-server interfaces between tasks is specified below.

Rule: Receives Only Stimuli From Multiple Sources (Third Preference)

if

$Task_{SENDER}$ sends a $Stimulus_{DF}$ to $Task_{RECEIVER}$ and
$Task_{RECEIVER}$ **receives only Stimuli from multiple sending tasks**

then

create a Queued $Message_{REQUEST}$ from $Task_{SENDER}$ to $Task_{RECEIVER}$
establish the design relationship $Task_{SENDER}$ Sends $Message_{REQUEST}$
establish the design relationship $Task_{RECEIVER}$ Receives $Message_{REQUEST}$
denote the traceability between $Stimulus_{DF}$ and $Message_{REQUEST}$
record the decision and rationale in the history for $Message_{REQUEST}$
create an appropriate Message $Data_{MD}$ for $Message_{REQUEST}$
establish the design relationship $Message_{REQUEST}$ Includes Message $Data_{MD}$
denote the traceability between $Stimulus_{DF}$ and Message $Data_{MD}$
record the decision and rationale in the history for Message $Data_{MD}$

fi

An example where this rule applies can be found in the elevator control system case study discussed previously. An internal task, Scheduler, sends a data flow, Scheduler Request, to another internal task, Elevator Manager. Elevator Manager receives only data flows from two tasks, Scheduler and Monitor Elevator Buttons. The rule defined above maps the data flow, Scheduler Request, to a queued message.

7.3.3 Rule for Mapping Ambiguous Data Flows

Ambiguous situations involving inter-task data flows are referred to an experienced designer to elicit additional information. Where an experienced designer is unavailable or unable to supply any additional insight, ambiguous data flows are mapped to queued messages. The rule for these situations is specified below.

Rule: Stimulus To Message (Last Preference)

if

$Task_{SENDER}$ sends a $Stimulus_{DF}$ to $Task_{RECEIVER}$

then

if an experienced designer is available

then show the designer all Stimuli from $Task_{SENDER}$ to $Task_{RECEIVER}$

ask the designer whether the sender must wait for the receiver to accept these Stimuli before continuing

if the designer says the sender need not wait or the designer does not know the answer

then create a Queued $Message_{DM}$ from $Task_{SENDER}$ to $Task_{RECEIVER}$

establish the design relationship $Task_{SENDER}$ Sends $Message_{DM}$

establish the design relationship $Task_{RECEIVER}$ Receives $Message_{DM}$

denote the traceability between $Stimulus_{DF}$ and $Message_{DM}$

record the decision and rationale in the history for $Message_{DM}$

else create a Tightly-Coupled $Message_{DM}$ from $Task_{SENDER}$ to $Task_{RECEIVER}$

establish the design relationship $Task_{SENDER}$ Sends $Message_{DM}$

establish the design relationship $Task_{RECEIVER}$ Receives $Message_{DM}$

denote the traceability between $Stimulus_{DF}$ and $Message_{DM}$

record the decision and rationale in the history for $Message_{DM}$

fi

else create a Queued $Message_{DM}$ from $Task_{SENDER}$ to $Task_{RECEIVER}$

establish the design relationship $Task_{SENDER}$ Sends $Message_{EM}$

establish the design relationship $Task_{RECEIVER}$ Receives $Message_{EM}$

denote the traceability between $Stimulus_{DF}$ and $Message_{DM}$

record the decision and rationale in the history for $Message_{DM}$

fi

fi

A situation where this rule applies can be found in the cruise control and monitoring system used in previous examples. In this case, a task, Auto Speed Control, sends three data flows, all called Throttle Value, to the task Throttle. No inference can be drawn about this task interface, so an experienced designer will be consulted, if possible; otherwise, these data flows are mapped to a queued message.

7.3.4 Rules for Mapping to Existing Messages

The design rules specified in this dissertation assume that all data flow exchanges between specific pairs of tasks occur in a uniform manner. So, for example, if a $task_A$ sends one data flow to another $task_B$ using a queued message, then all data flows from $task_A$ to $task_B$ will be sent using a queued message. This assumption allows specific inferences to be drawn whenever a mechanism exists to carry any data flow between a pair of tasks. Specifically, once a message interface is defined to carry a data flow from a source task to a destination task then any remaining data flows with the same source and destination can be mapped to the same message interface.

Four rules are specified to perform these mappings. One rule maps data flows onto an existing queued message between two tasks, provided that the existing queued message carries at least one data flow. The second rule maps data flows onto an existing tightly-coupled message between two tasks, provided that the existing tightly-coupled message carries at least one data flow. The third rule maps responses onto an existing tightly-coupled message between two tasks, provided that the existing tightly-coupled message carries at least one response. The fourth rule maps data flows from a source task

to triggered synchronous functions within a control task onto the same message from that source task that carries events destined for the control task. The rule to map data flows onto an existing queued message is specified below.

Rule: Stimulus Rides On Existing Queued Message (First Preference)

if
$Task_{SENDER}$ sends a $Stimulus_{DF}$ to $Task_{RECEIVER}$ and
$Task_{SENDER}$ sends a Queued $Message_{QM}$ to $Task_{RECEIVER}$ and
$Message_{QM}$ is derived from a Stimulus
then
if an appropriate Message $Data_{MD}$ does not exist for $Message_{QM}$
then create an appropriate Message $Data_{MD}$ for $Message_{QM}$
establish the design relationship $Message_{QM}$ Includes Message $Data_{MD}$
record the decision and rationale in the history for Message $Data_{MD}$
else use an existing Message $Data_{MD}$ for $Message_{QM}$
fi
denote the traceability between $Stimulus_{DF}$ and Message $Data_{MD}$
record the decision and rationale in the history for Message $Data_{MD}$
if the traceability between $Stimulus_{DF}$ and $Message_{QM}$ is not already noted
then denote the traceability between $Stimulus_{DF}$ and $Message_{QM}$
record the decision and rationale in the history for $Message_{QM}$
fi
fi

The application of this rule can best be illustrated together with the application of a similar rule, specified below, that maps a data flow onto an existing tightly-coupled message that already carries a data flow between two tasks.

Rule: Stimulus Rides On Existing Tightly-Coupled Message (First Preference)

if

$Task_{SENDER}$ sends a $Stimulus_{DF}$ to $Task_{RECEIVER}$ and
$Task_{SENDER}$ sends a Tightly-Coupled $Message_{TCM}$ to $Task_{RECEIVER}$ and
$Message_{TCM}$ is derived from a Stimulus

then

if an appropriate Message $Data_{MD}$ does not exist for $Message_{TCM}$
then create an appropriate Message $Data_{MD}$ for $Message_{TCM}$
establish the design relationship $Message_{TCM}$ Includes Message $Data_{MD}$
record the decision and rationale in the history for Message $Data_{MD}$
else use an existing Message $Data_{MD}$ for $Message_{QM}$
fi
denote the traceability between $Stimulus_{DF}$ and Message $Data_{MD}$
record the decision and rationale in the history for Message $Data_{MD}$
if the traceability between $Stimulus_{DF}$ and $Message_{TCM}$ is not already noted
then denote the traceability between $Stimulus_{DF}$ and $Message_{TCM}$
record the decision and rationale in the history for $Message_{TCM}$
fi

fi

An example occurs in the cruise control and monitoring system, discussed previously, where either of the two rules defined above might apply. In the example, the task Auto Speed Control sends three data flows, all named Throttle Value, to the task Throttle. The type of interface needed between these two tasks might be determined after consulting an experienced designer or by a default decision. The interface could be mapped to either a queued message or a tightly-coupled message. If one of the three data flows is mapped to a queued message, then the first of the two rules specified above would map the remaining two data flows to the same message. If, on the other hand, one of the data flows becomes mapped to a tightly-coupled message, then the second of the

two rules would map the remaining two data flows to the same tightly-coupled message. A similar rule, specified below, exists to handle redundant responses between tasks.

Rule: Response Rides On Existing Tightly-Coupled Message (First Preference)

if
$Task_{SENDER}$ sends a $Response_{R}$ to $Task_{RECEIVER}$ and
$Task_{SENDER}$ sends a Tightly-Coupled $Message_{TCM}$ to $Task_{RECEIVER}$ and
$Message_{TCM}$ is derived from a Response
then
if an appropriate Message $Data_{MD}$ does not exist for $Message_{TCM}$
then create an appropriate Message $Data_{MD}$ for $Message_{TCM}$
establish the design relationship $Message_{TCM}$ Includes Message $Data_{MD}$
record the decision and rationale in the history for Message $Data_{MD}$
else use an existing Message $Data_{MD}$ for $Message_{QM}$
fi
denote the traceability between $Response_{R}$ and Message $Data_{MD}$
record the decision and rationale in the history for Message $Data_{MD}$
if the traceability between $Response_{R}$ and $Message_{TCM}$ is not already noted
then denote the traceability between $Response_{R}$ and $Message_{TCM}$
record the decision and rationale in the history for $Message_{TCM}$
fi
fi

A special circumstance exists when a task sends event flows to a control task and also sends data flows to that control task. Since control objects cannot accept data flows, data is sometimes sent to a control task as a data flow to a synchronous function triggered by the control object. In such situations, the data flow from the sending task should ride on the same message as the event flows from the sending task. The rule specified below recognizes and acts upon such situations.

Rule: Stimulus To Control Task Rides With Events (Second Preference)

if

$Task_{SENDER}$ sends a $Stimulus_{DF}$ to $Task_{RECEIVER}$ and
$Task_{SENDER}$ sends a $Message_{EM}$ to $Task_{RECEIVER}$ and
the sink for $Stimulus_{DF}$ is a Triggered Synchronous Function

then

create an appropriate Message $Data_{MD}$ for $Message_{EM}$
establish the design relationship $Message_{EM}$ Includes Message $Data_{MD}$
denote the traceability between $Stimulus_{DF}$ and Message $Data_{MD}$
record the decision and rationale in the history for Message $Data_{MD}$
denote the traceability between $Stimulus_{DF}$ and $Message_{EM}$
record the decision and rationale in the history for $Message_{EM}$

fi

An example where this rule applies can be found in the robot controller case study used previously. In the example, a data flow, Program Number, from the task Control Panel Input Handler goes to a triggered synchronous function, Change Program, within the control task Robot Command Processor. The rule specified above maps the data flow, Program Number, to the message, defined earlier by another rule, that carries the events from the task Control Panel Input Handler to the control task Robot Command Processor.

7.3.5 Rule for Symmetric Message Interfaces

A special rule is defined based upon the assumption that data flows exchanged between a pair of tasks, where the data flow in one direction is already allocated to a message, can typically be mapped to a symmetric interface. This assumption means that if data flows in one direction are mapped to a queued message, then data flows in the reverse direction can also be mapped to a queued message. Similarly, if data flows in one direction are mapped to a tightly-coupled message, then data flows in the reverse

direction can also be mapped to a tightly-coupled message. Asymmetric message interfaces with respect to data flows are assumed then to apply only when one data flow is sent in response to another. The rule to map a data flow in one direction to the same type of message as exists in the reverse direction is specified below.

Rule: Stimulus For Reverse Channel (Second Preference)

if

$Task_{SENDER}$ sends a $Stimulus_{DF}$ to $Task_{RECEIVER}$ and
$Task_{RECEIVER}$ sends a $Message_{RtoS}$ to $Task_{SENDER}$ and
$Message_{RtoS}$ is derived from an Internal Data Flow

then

determine the type of $Message_{RtoS}$ and then create a $Message_{StoR}$ of
the same type from $Task_{SENDER}$ to $Task_{RECEIVER}$
establish the design relationship $Task_{SENDER}$ Sends $Message_{StoR}$
establish the design relationship $Task_{RECEIVER}$ Receives $Message_{StoR}$
denote the traceability between $Stimulus_{DF}$ and $Message_{StoR}$
record the decision and rationale in the history for $Message_{StoR}$
create an appropriate Message $Data_{MD}$ for $Message_{REQUEST}$
establish the design relationship $Message_{REQUEST}$ Includes Message $Data_{MD}$
denote the traceability between $Stimulus_{DF}$ and Message $Data_{MD}$
record the decision and rationale in the history for Message $Data_{MD}$

fi

An example where this rule applies can be found in the robot controller case study discussed previously. In the example, the data flow Motion Blocks is sent from the task Interpreter to the task Axis Manager and the data flow Motion Acknowledgments is sent from the task Axis Manager to the task Interpreter. These data flows are sent independently of each other. Assuming that the data flow Motion Blocks becomes mapped to a queued message then the rule specified above will also map the data flow

Motion Acknowledgments to a queued message going in the opposite direction. This mapping occurs without consulting the designer.

7.3.6 Rule for Ambiguous Interface to Input/Output Devices

An ambiguity can arise when a task both sends and receives a tightly-coupled message to a device input/output task. Typically, one of these tightly-coupled messages is sent in response to the other; however, in cases where neither of the tightly-coupled messages is derived from a data flow sent in response to another data flow, no means exist to infer which, if either, tightly-coupled message is the request and which is the response. A rule is defined to recognize such situations and to refer them to an experienced designer for a determination. If no experienced designer is available, or if the experienced designer is unsure about the relationship between the messages, then the message flowing from the device input/output task is mapped, by default, as an answer to the message flowing to the device input/output task. This default mapping is considered most likely to apply in situations involving interfaces to device input/output tasks.

An example where this situation might arise can be found in the robot controller case study. In the example, the device input/output task receives a data flow, Axis Block, from the internal task Axis Manager and sends a data flow, Axis Acknowledgment, to Axis Manager. Assume that the designer chooses to map Axis Block onto a tightly-coupled message, and that the symmetric interface rule mapped Axis Acknowledgment onto a tightly-coupled message. The rule specified below would detect this possible ambiguity and either consult an experienced designer or make a default

decision that the message carrying Axis Acknowledgment answers the message carrying Axis Block.

Rule: Finding Request And Response (Last Preference)

if

$Task_{IO}$ is a periodic or an asynchronous device input/output task and
$Task_{ANY}$ is any task other than $Task_{IO}$ and
Tightly-Coupled $Message_{OUT}$ flows from $Task_{ANY}$ to $Task_{IO}$ and
Tightly-Coupled $Message_{IN}$ flows from $Task_{IO}$ to $Task_{ANY}$

then

if an experienced designer is available
then ask the designer whether:
1) $Message_{IN}$ Answers $Message_{OUT}$ or
2) $Message_{OUT}$ Answers $Message_{IN}$ or
3) Neither Answers the other or
4) the designer is unsure about the relationship
if the designer chooses the first or fourth options
then establish the design relationship $Message_{IN}$ Answers $Message_{OUT}$
record the decision and rationale in the history for $Message_{IN}$
else if the designer chooses option two
then establish the design relationship $Message_{OUT}$ Answers $Message_{IN}$
record the decision and rationale in the history for $Message_{OUT}$
fi
fi
else establish the design relationship $Message_{IN}$ Answers $Message_{OUT}$
record the decision and rationale in the history for $Message_{IN}$
fi
fi

7.4 Elicit Message Priorities

When a task receives queued messages from multiple sources a designer might wish to give priorities to the inputs from each source based upon requirements of the application. Since no means exist to infer these priorities, an experienced designer must be consulted to elicit them. The rule specified below detects situations where such elicitation might be appropriate and then manages the elicitation process.

Rule: User Specifies Queued Message Priority

if
an experienced designer is available and
any task receives queued messages from multiple sending tasks
then
offer the designer a chance to review and optionally to alter the priority of queued messages received by each task that receives queued messages from multiple sending tasks
if the designer accepts the offer
then **for each** task in the design that receives queued messages from multiple senders
show the designer the Set_{QM} of queued messages received by the task, as well as the priority of each message
offer the designer a chance to assign a new priority to any of the messages in the Set_{QM}
update the priority of each member of Set_{QM} as indicated by the designer
rof
fi
fi

Regarding the case studies used to illustrate previous rules, both the cruise control and monitoring system and the elevator control system lead to task interfaces where differing priorities might be assigned to queued messages. In the former case, the task Cruise Control receives queued messages from three other tasks, Monitor Auto Sensors,

Monitor Cruise Control Inputs, and Auto Speed Control. An experienced designer might choose to assign a higher priority to messages from Monitor Auto Sensors than to messages from the other input tasks. In the case of the elevator control system, the task Elevator Manager receives queued messages from two tasks, Scheduler and Monitor Elevator Buttons. An experienced designer might choose to assign greater precedence to messages received from the Scheduler task.

7.5 Allocate Queue Interfaces

The fifth decision-making process, Allocate Queue Interfaces, compares the requirements for queued messages in the evolving design with the message queuing facilities available in the target environment. The issues considered during this decision-making process occur within environments, such as the Ada run-time system, where message queuing facilities are not available. More generally, one can imagine target environments that support only single-priority message queues, those that support only multiple-priority message queues, those that support both, and those that support neither. Design-decision rules are specified for each of the relevant situations. Guidance for handling environments where no message queuing facilities exist is taken from the Ada-based Design Approach for Real-Time Systems (ADARTS) described by Gomaa. [Gomaa93, Chapter 17] In such cases, an intermediary queue-control task is constructed to manage an internal message list. The interfaces between the intermediary task and the message senders and receivers is mapped to tightly-coupled messages.

7.5.1 Rules for Single-Priority Message Queues

Three rules are defined to consider cases where a task receives queued messages at only one priority. The first of these rules recognizes when message queues are available in the target environment and then creates an incoming queue for the receiving task and maps each queued message received by the task into the newly created queue. The rule is specified below.

Rule: Single Priority Message Queue Available

if
  $Task_{RECEIVER}$ receives queued messages at a single priority and
  message queues are available in the target environment
then
  create a $Queue_{RQ}$ for $Task_{RECEIVER}$
  establish the design relationship $Task_{RECEIVER}$ Consumes $Queue_{RQ}$
  record the decision and rationale in the history for $Queue_{RQ}$
  for each Queued $Message_{QM}$ to $Task_{RECEIVER}$
    establish the design relationship $Queue_{RQ}$ Holds $Message_{QM}$
    record the decision and rationale in the history for $Queue_{RQ}$
  rof
fi

This rule applies in all of Gomaa's case studies, discussed to this point. As described by Gomaa, the case studies each have at least one message queue, all queued messages arrive at a single priority, and message queuing is available in the target environment.

The second rule recognizes that a single priority message queue is required but that only priority message queuing services are supported by the target environment. In

such cases, the rule creates a priority queue for the receiving task and then maps each incoming message into a single priority slot in the queue. The effect simulates a single-priority message queue. The rule is specified below.

Rule: Single Priority Only Priority Queues Available

if

$Task_{RECEIVER}$ receives queued messages at a single priority and
priority queues are available in the target environment and
message queues are unavailable in the target environment

then

create a Priority $Queue_{PQ}$ for $Task_{RECEIVER}$
establish the design relationship $Task_{RECEIVER}$ Owns $Queue_{PQ}$
record the decision and rationale in the history for $Queue_{PQ}$
create a $Queue_{SQ}$ as a subqueue for $Queue_{PQ}$
establish the design relationship $Queue_{PQ}$ Heads $Queue_{SQ}$
record the decision and rationale in the history for $Queue_{SQ}$
for each Queued $Message_{QM}$ to $Task_{RECEIVER}$
establish the design relationship $Queue_{SQ}$ Holds $Message_{QM}$
record the decision and rationale in the history for $Queue_{SQ}$
rof

fi

A third rule recognizes that message queues are required for a task but that no message queuing services of any kind are available in the target environment. In such cases, the rule creates a queue-control task for the receiving task and encapsulates a queue within the queue-control task. A tightly-coupled message is defined from the receiving task to the queue-control task in order to request the next available message from the internal queue of the queue-control task. A tightly-coupled message is defined from the queue-control task to the receiving task to return the next available message to the

receiving task. When no message is available in the internal queue of the queue-control task then the receiving task blocks until a message is available. The rule is specified below.

Rule: Single Priority No Queues Available

if

$Task_{RECEIVER}$ receives queued messages at a single priority and
priority queues are unavailable in the target environment and
message queues are unavailable in the target environment

then

create a queue control $Task_{QC}$ for $Task_{RECEIVER}$
record the decision and rationale in the design history for $Task_{QC}$
create a $Queue_{RQ}$ for $Task_{RECEIVER}$
establish the design relationship $Task_{QC}$ Encapsulates $Queue_{RQ}$
record the decision and design rationale in the design history for $Task_{QC}$
create a Tightly-Coupled $Message_{RFNM}$ from $Task_{RECEIVER}$ to $Task_{QC}$
establish the design relationship $Task_{RECEIVER}$ Sends $Message_{RFNM}$
establish the design relationship $Task_{QC}$ Receives $Message_{RFNM}$
record the decision and rationale in the history for $Message_{RFNM}$
create a Tightly-Coupled $Message_{NMA}$ from $Task_{QC}$ to $Task_{RECEIVER}$
establish the design relationship $Task_{QC}$ Sends $Message_{NMA}$
establish the design relationship $Task_{RECEIVER}$ Receives $Message_{NMA}$
record the decision and rationale in the history for $Message_{NMA}$
establish the design relationship $Message_{NMA}$ Answers $Message_{RFNM}$
for each Queued $Message_{QM}$ received by $Task_{RECEIVER}$
establish the design relationship $Queue_{RQ}$ Holds $Message_{QM}$
record the decision and rationale in the history for $Queue_{RQ}$
establish the design relationship $Message_{NMA}$ Carries $Message_{QM}$
record the decision and rationale in the history for $Message_{NMA}$
create a Tightly-Coupled $Message_{SUBMIT}$ from the TaskSENDER,
where $Task_{SENDER}$ sends $Message_{QM}$ to $Task_{QC}$
establish the design relationship $Message_{SUBMIT}$ Carries $Message_{QM}$
record the decision and rationale in the history for $Message_{SUBMIT}$
establish the design relationship $Task_{SENDER}$ Sends $Message_{SUBMIT}$
establish the design relationship $Task_{QC}$ Receives $Message_{SUBMIT}$
remove all design relationships Send and Receive involving
$Message_{QM}$
rof

fi

Note also that a tightly-coupled message is created from each sending task to the queue-control task. These tightly-coupled messages are used to submit queued messages to the queue-control task for later delivery to the receiving task. Each queued message to be delivered is carried within a tightly-coupled message.

7.5.2 Rules For Multiple-Priority Message Queues

Three additional rules are defined to consider cases where a task receives queued messages at multiple priorities. The first of these rules, specified below, recognizes when priority message queues are available in the target environment and then creates an incoming priority queue for the receiving task.

Rule: Multiple Priority Priority Queues Available

if

$Task_{RECEIVER}$ receives queued messages at multiple priorities and
priority queues are available in the target environment

then

create a Priority $Queue_{PQ}$ for $Task_{RECEIVER}$
establish the design relationship $Task_{RECEIVER}$ Owns $Queue_{PQ}$
record the design decision and rationale in the design history for $Queue_{PQ}$
construct the $Set_{PRIORITIES}$ of priorities at which $Task_{RECEIVER}$ receives
    queued messages
for each $Priority_P$ in $Set_{PRIORITIES}$
    create a $Queue_{SQ}$ as a subqueue for $Queue_{PQ}$
    establish the design relationship $Queue_{PQ}$ Heads $Queue_{SQ}$
    record the decision and rationale in the history for $Queue_{PQ}$
    for each Queued $Message_{QM}$ with $Priority_P$ to $Task_{RECEIVER}$
        establish the design relationship $Queue_{SQ}$ Holds $Message_{QM}$
        record the decision and rationale in the history for $Queue_{SQ}$
    rof
rof

fi

A subqueue is created for each priority at which queued messages are received by the receiving task. Incoming queued messages are mapped into the appropriate subqueue based on priority.

A second rule, specified below, recognizes that priority queues are not available in the target environment but that single-priority message queues are available.

Rule: Multiple Priority Only Message Queues Available

if

$Task_{RECEIVER}$ receives queued messages at multiple priorities and
priority queues are unavailable in the target environment and
message queues are available in the target environment

then

construct the $Set_{PRIORITIES}$ of priorities at which $Task_{RECEIVER}$ receives queued messages
for each $Priority_P$ in $Set_{PRIORITIES}$
create a $Queue_{RQ}$ for $Task_{RECEIVER}$
establish the design relationship $Task_{RECEIVER}$ Consumes $Queue_{RQ}$
record the decision and rationale in the history for $Queue_{RQ}$
for each Queued $Message_{QM}$ with $Priority_P$ to $Task_{RECEIVER}$
establish the design relationship $Queue_{RQ}$ Holds $Message_{QM}$
record the decision and rationale in the history for $Queue_{RQ}$
rof
rof
fi

This rule simulates priority queues by creating a message queue for each priority at which queued messages arrive at the receiving task. Incoming queued messages are mapped to the appropriate queue based on priority.

A third rule recognizes that neither priority queues nor single-priority message queues are available in the target environment. This rule is specified below.

Rule: Multiple Priority No Queues Available

if

$Task_{RECEIVER}$ receives queued messages at multiple priorities and
priority queues are unavailable in the target environment and
message queues are unavailable in the target environment

then

create a queue control $Task_{QC}$ for $Task_{RECEIVER}$
record the decision and rationale in the design history for $Task_{QC}$
create a Priority $Queue_{PQ}$ for $Task_{RECEIVER}$
establish the design relationship $Task_{QC}$ Encloses $Queue_{RQ}$
record the decision and design rationale in the history for $Task_{QC}$
create a Tightly-Coupled $Message_{RFNM}$ from $Task_{RECEIVER}$ to $Task_{QC}$
establish the design relationship $Task_{RECEIVER}$ Sends $Message_{RFNM}$
establish the design relationship $Task_{QC}$ Receives $Message_{RFNM}$
record the decision and design rationale in the history for $Message_{RFNM}$
create a Tightly-Coupled $Message_{NM}$ from $Task_{QC}$ to $Task_{RECEIVER}$
establish the design relationship $Task_{QC}$ Sends $Message_{NM}$
establish the design relationship $Task_{RECEIVER}$ Receives $Message_{NM}$
record the decision and design rationale in the history for $Message_{NM}$
establish the design relationship $Message_{NM}$ Answers $Message_{RFNM}$
construct the $Set_{PRIORITIES}$ of priorities at which $Task_{RECEIVER}$ receives
queued messages
for each $Priority_P$ in $Set_{PRIORITIES}$
create a $Queue_{SQ}$ as a subqueue for $Queue_{PQ}$
establish the design relationship $Queue_{PQ}$ Heads $Queue_{SQ}$
record the decision and design rationale in the history for $Queue_{PQ}$
for each Queued $Message_{QM}$ received by $Task_{RECEIVER}$
establish the design relationship $Queue_{SQ}$ Holds $Message_{QM}$
record the decision and design rationale in the history for $Queue_{SQ}$
establish the design relationship $Message_{NM}$ Carries $Message_{QM}$
record the decision and design rationale in the history for $Message_{NM}$
create a Tightly-Coupled $Message_{SUB}$ from the TaskS, where $Task_S$
sends $Message_{QM}$ to $Task_{QC}$
establish the design relationship $Message_{SUB}$ Carries $Message_{QM}$
record the decision and design rationale in the history for $Message_{SUB}$
establish the design relationship $Task_{SENDER}$ Sends $Message_{SUB}$
establish the design relationship $Task_{QC}$ Receives $Message_{SUB}$
remove all design Sends and Receives involving $Message_{QM}$
rof
rof

fi

This rule creates a queue-control task for the receiving task and encloses a priority queue within the queue-control task. A tightly-coupled message is defined from the receiving task to the queue-control task in order to request the next available message from the internal queue of the queue-control task. A tightly-coupled message is defined from the queue-control task to the receiving task to return the next available message to the receiving task. When messages are available at multiple priorities, the queue-control task returns the oldest message of the highest priority. When no message is available in the internal queue of the queue-control task then the receiving task blocks until a message is available. Note also that a tightly-coupled message is created from each sending task to the queue-control task. These tightly-coupled messages are used to submit queued messages to the queue-control task for later delivery to the receiving task. Each queued message to be delivered is carried within a tightly-coupled message.

7.6 Consider Renaming Task Interface Design Elements

The final decision-making process, among those contained within the Task Interface Definition Knowledge base, offers the designer an opportunity to review the new design elements created. If the designer is dissatisfied with the results, then the results can be discarded. In addition, the designer is given an opportunity to rename design elements, which might include: queue-control tasks, single-priority and multiple-priority message queues, queued and tightly-coupled messages, and message data fields. A single rule, not shown here, drives the review and renaming that completes the definition of task interfaces.